RANCANG BANGUN SISTEM E-LEARNING PEMROGRAMAN PADA MODUL DETEKSI PLAGIARISME KODE PROGRAM DAN *STUDENT FEEDBACK SYSTEM*

Abdul Munif[1)], Rizky Januar Akbar[2)], Ruchi Intan Tantra[3)], Rachmania Ilavi[4)]
[1, 2, 3, 4)]Jurusan Teknik Informatika, Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Jl. Teknik Kimia, Gedung Teknik Informatika,Kampus ITS Sukolilo, Surabaya, 60111
e-mail: munif@if.its.ac.id[1)]

ABSTRAK

Kompetensi utama yang harus dimiliki oleh mahasiswa jurusan ilmu komputer/informatika adalah pemrograman. Perkuliahan yang berbasis pemrograman seringkali mewajibkan mahasiswa untuk mengerjakan soal yang cukup banyak. Hal ini tentunya akan menyulitkan dosen dalam mengevaluasi hasil pekerjaan mahasiswa. Selain itu, dimungkinkan pula adanya praktik mahasiswa yang memplagiat hasil dari mahasiswa lain. Penelitian ini bertujuan untuk menjawab permasalahan tersebut. Pada penelitian ini dikembangkan sebuah sistem pembelajaran/e-learning pemrograman.

Modul sistem e-learning yang dibuat pada artikel ini terbatas pada modul deteksi kemiripan kode program dan student feedback system. Modul deteksi kemiripan kode program berfungsi untuk mengecek kemiripan kode program antar mahasiswa. Kemudian kode-kode program yang memiliki tingkat kemiripan tinggi akan dikelompokkan menjadi satu menggunakan algoritma hierarchical clustering. Proses pengecekan kemiripan program dimulai dari proses transformasi kode program ke dalam Abstract Syntax Tree (AST), kemudian ditransformasi menjadi sequence dan dihitung kemiripannya menggunakan algoritma Levenshtein Distance.

Modul student feedback system berfungsi untuk mengecek kemiripan kode program mahasiswa dengan dosen. Mahasiswa akan mendapatkan informasi apakah kode program mereka sudah sesuai dengan yang diinginkan oleh dosen atau belum. Sebelum sistem memberikan umpan balik, terlebih dahulu kode program dosen dan mahasiswa diproses menjadi AST dan kemudian menjadi sequence. Sistem akan membandingkan kemiripan kode program dosen dan mahasiswa menggunakan algoritma Smith-Waterman yang telah dimodifikasi. Kemudian sistem menampilkan baris-baris kode mana saja yang sama antara kode program dosen dengan kode program mahasiswa. Dari hasil pengujian dapat disimpulkan bahwa modul deteksi plagiarisme dan student feedback system telah terimplementasi dengan baik.

***Kata Kunci**: abstract syntax tree, clustering, plagiarisme, student feedback system.*

ABSTRACT

Programming is one of key competencies that must be owned by students majoring in computer science. Programming course often require students to work on a lot of computational problems. Due to a lot evaluation of students' work result needs to be done, lecturers sometime feel overwhelmed. It is also possible that students have plagiarize other student's works and it is hard to prove. This study aims to answer these problems by developing an e-learning system specialized in programming course.

This article covers certain part of e-learning system that will be build, plagiarism detection module and student feedback systems. The main function of plagiarism detection module is to check the similarities between students' source code. The source codes that have high degree of similarity are group together using hierarchical clustering. The program begins the process by transforming source code into Abstract Syntax Tree (AST), and then transformed into a sequence. This sequence then used to calculate its similarity using Levenshtein Distance algorithm.

Student feedback system checks the similarity between students' code and lecturer's code. The system inform the students whether their program code is corresponding with lecturer's code or not. Before comparing the students' and lecturer's code, this module will convert their source code into AST and sequences. The similarity is calculated by using Smith-Waterman algorithm. The system then displays students' code that corresponding with lecturer's code. Based on experiment results, we can conclude that plagiarism detection module and student feedback system have been succesfully implemented.

***Keywords**: abstract syntax tree, clustering, plagiarism, student feedback system.*

I. PENDAHULUAN

Salah satu kompetensi yang harus dimiliki oleh mahasiswa jurusan ilmu komputer atau informatika adalah pemrograman. Beragam mata kuliah yang diajarkan mewajibkan mahasiswa untuk dapat membuat program dalam menyelesaikan permasalahan komputasional. Hal ini menuntut banyaknya tugas pemrograman yang harus diberikan agar mahasiswa semakin mahir. Namun, banyaknya tugas pemrograman membuat pengajar mengalami kesulitan dalam melakukan penilaian. Selain itu, seringkali ditemukan banyak mahasiswa yang memplagiat hasil pekerjaan mahasiswa lain. Hal ini semakin mempersulit pengajar memberikan penilaian yang objektif.

Salah satu kakas bantu untuk mengecek kebenaran kode program jawaban mahasiswa adalah *online judge*. Pengajar dapat menggunakan *online judge* (contoh: SPOJ [1]) dengan cara menentukan data input dan output dari sebuah permasalahan komputasi. Kemudian kode program mahasiswa dikompilasi, dijalankan dengan mengambil input acak yang sudah didefinisikan, dan kemudian mengecek output program. Apabila hasil output program sama dengan output yang sudah didefinisikan sebelumnya, maka dianggap bahwa kode program jawaban mahasiswa tersebut benar.

Namun, selain mengecek kebenaran output program pengajar juga harus mengecek orisinalitas kode program mahasiswa. Pada kenyataannya banyak ditemukan kode program dari beberapa mahasiswa yang memiliki tingkat kemiripan yang tinggi. Kode-kode program tersebut dianggap mirip biasanya karena hanya terdapat penggantian nama variabel, perubahan letak blok program, atau mengubah klausa perulangan dan percabangan. Namun, kemiripan tersebut tidak bisa langsung dianggap sebagai salah bentuk plagiarisme. Ada kalanya karena soal komputasi tersebut cukup sederhana, maka kemungkinan jawaban mahasiswa akan seragam.

Pencocokan kode-kode program yang mirip tersebut tidak dapat menggunakan metode pencocokan teks biasa. Hal ini disebabkan karena perbedaan karakteristik dokumen teks biasa dengan kode program. Salah satu contoh metode yang dapat digunakan untuk mengecek kemiripan struktur kode program adalah dengan menggunakan *Abstract Syntax Tree* (AST) [2]. Kode program yang akan dianalisis terlebih dahulu dibentuk AST-nya menggunakan kakas bantu ANTLR (*Another Tool for Language Recognition*). Setelah AST setiap kode program dibentuk, maka langkah selanjutnya adalah membandingkan AST tiap kode program. Ada beberapa metode yang digunakan di dalam artikel ini, yaitu algoritma Smith-Waterman [3] yang dimodifikasi, dan algoritma Levenshtein Distance [4].

Pengecekan kemiripan kode program ternyata tidak hanya bermanfaat untuk mendeteksi plagiarisme saja. Ada manfaat lain yang bisa kita peroleh untuk meningkatkan kualitas pembelajaran. Salah satunya adalah pengecekan kemiripan kode program mahasiswa dengan kode program jawaban dari dosen. Dengan adanya pengecekan kemiripan ini, maka kita dapat melihat apakah mahasiswa telah dapat membuat kode program seperti yang diharapkan oleh pengajar. Apabila kode program mahasiswa belum sesuai, maka dapat diberikan evaluasi bagian mana saja dari kode program mahasiswa yang belum sesuai.

Penelitian ini berfokus pada dua hal tersebut di atas, yaitu: (1) pendeteksian kemiripan kode program antar mahasiswa untuk mendeteksi plagiarisme, dan (2) membuat sistem umpan balik (*feedback*) bagi mahasiswa. Kedua hal tersebut merupakan bagian dari sebuah sistem pembelajaran daring (*e-learning*) pemrograman yang akan dibangun. Dengan adanya sistem ini diharapkan proses evaluasi pembelajaran dapat dilakukan dengan lebih mudah. Artikel ini akan menjelaskan lebih detail mengenai metode-metode yang digunakan, rancangan sistem, pengujian, dan evaluasi, serta kesimpulan yang didapat.

II. TINJAUAN PUSTAKA

A. ANTLR

ANTLR (*Another Tool for Language Recognition*) adalah sebuah *parser generator* yang dapat digunakan untuk membaca, memproses, dan mengeksekusi, atau mengartikan struktur teks atau file binari. Dari *grammar* (aturan bahasa), ANTLR dapat menghasilkan *lexer, parser*, *parser trees* dan kombinasi *lexer-parser* [5]. Parser otomatis dapat menghasilkan *Abstract Syntax Tree* (AST) yang dapat diproses lebih lanjut dengan *tree parser*.

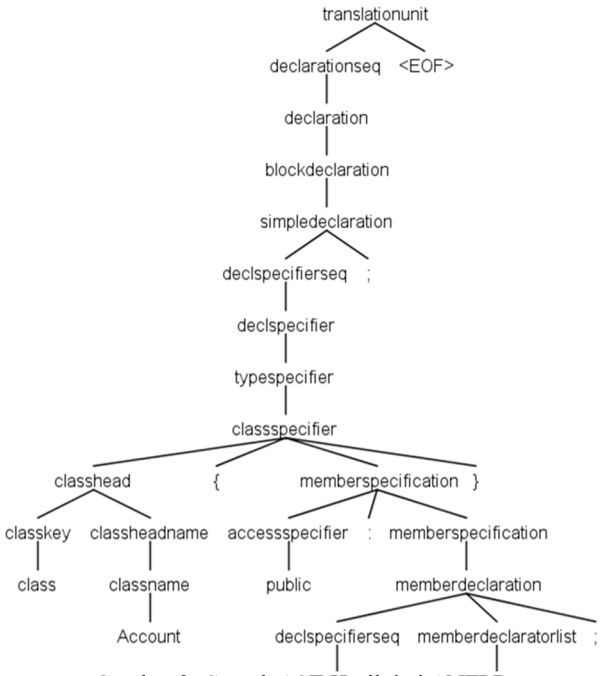
ANTLR menyediakan alat bantu untuk mekanisme pembacaan AST yang ada pada *runtime library* ANTLR, yaitu *listener* dan *visitor*. Mekanisme yang digunakan untuk mengecek tiap elemen dari AST adalah *tree walking* secara *Depth First Search* (DFS). Fungsi *listener* lebih mudah digunakan namun kurang fleksibel karena tidak dapat mengubah alur eksekusi. Sedangkan fungsi *visitor* dapat mengatur atau mengubah suatu alur eksekusi sesuai dengan yang diinginkan. Secara umum, *listener* lebih tepat dipakai untuk membaca sebuah dokumen dengan alur yang pasti. Fungsi *listener* memiliki dua buah metode yaitu `enter` dan `exit`. Sedangkan *visitor* hanya memiliki satu metode yaitu `visit` [6]. ANTLR menggunakan input yaitu file *grammar* yang mendeskripsikan suatu aturan dalam sebuah bahasa/dokumen. Untuk pembacaan kode program C++, ANTLR telah menyediakan *grammar* khusus yang dapat diakses di repositori ANTLR [7].

```
#include <cstudio>
#include <cstdlib>
#include <iostream>

using namespace std;

class Account
{
  public:
    int balance;
};
```

Gambar 1. Contoh Kode Program



Gambar 2. Contoh AST Hasil dari ANTLR

B. Abstract Syntax Tree (AST)

Abstract Syntax Tree (AST) adalah sebuah pohon yang merepresentasikan struktur dari sebuah kode program dengan bahasa pemrograman tertentu. Dalam penelitian ini, AST diperoleh dari *tree parser* yang telah dihasilkan oleh ANTLR. Kode program akan diubah secara otomatis oleh ANTLR ke dalam bentuk AST. Dari AST inilah akan diproses lebih lanjut untuk mendeteksi kemiripan antar kode program (kode program sesama mahasiswa dan kode program mahasiswa dengan dosen). Sebagai contoh, pada Gambar 1 ditunjukkan kode program sederhana dalam bahasa C++. Kemudian kode program tersebut diproses menggunakan ANTLR untuk mendapatkan AST seperti yang dicontohkan pada Gambar 2.

Apabila diperhatikan, struktur AST yang dihasilkan cukup kompleks untuk kode program yang sederhana tersebut. Beberapa elemen AST yang penting pada Gambar 2 antara lain: *declaration*, *classspecifier*, *memberspecification*, *memberdeclaration*, dan sebagainya. Penjelasan lebih detil mengenai elemen-elemen AST ini akan dijelaskan pada tahap praproses data.

C. Algoritma Smith-Waterman

Algoritma Smith-Waterman dapat digunakan untuk menghitung kemiripan dua buah dokumen, tak terkecuali sebuah kode program. Algoritma ini digunakan terutama dalam bidang biologi komputasional, algoritma ini mempunyai efek yang bagus dalam pencocokan yang optimal. Pertama, algoritma ini akan menggunakan matriks, lalu menghasilkan sub matriks yang berisi semua kemungkinan kesamaan nilai dengan metode iteratif, membandingkan nilai-nilai dari sub matriks hingga didapatkan nilai yang optimal (didapatkan nilai tertinggi) [3].

Untuk dua buah AST *sequence*: $S = s_1, s_2, s_3, ..., s_n$ dan $T = t_1, t_2, t_3, ..., t_m$, menurut algoritma *dynamic*

```
 i = LengthS, j = LengthT

 SmithWaterman(S, T)
1. FOR i > 0 AND j > 0
2.   IF (D[i,j ]= D[i-1, j-1] + f(i,j) THEN
3.     i--
4.     j--
5.   ELSE IF (D[i,j] = D[i-1, j] + Wx)  THEN
6.     i--
7.   ELSE IF (D[i,j] = D[i-1, j-1] + Wy) THEN
8.     j--;
```

Gambar 3. *Pseudocode* Algoritma Smith-Waterman

programming, dibutuhkan sebuah matriks berukuran $(n+1)\times(m+1)$ untuk menampung hasil perhitungan jarak. Penghitungan matriks dan sub matriks dapat menggunakan (1) dan (2). Nilai indeks i dan j adalah $1 \le i \le n$ dan $1 \le j \le m$. Sedangkan sub matriks D dapat dihitung menggunakan *pseudocode* pada Gambar 3.

Perhitungan algoritma Smith-Waterman adalah sebagai berikut.

1. Kondisi awal

$$D(i,0) = D(0,j) = 0 \tag{1}$$

2. Relasi rekursi

$$D(i,j) = \max \begin{cases} 0 \\ D(i-1,j-1) + f(i,j) \\ \max\{D(i-1,j) - W_x\} \\ \max\{D(i,j-1) - W_y\} \end{cases} \tag{2}$$

Untuk $1 \le i \le n$ dan $1 \le j \le m$, W_x adalah kolom *penalty* dengan panjang x yang ditambahkan pada *sequence*, sedangkan W_y adalah kolom *penalty* dengan panjang y yang ditambahkan pada *sequence*, dan *f(i, j)* bernilai *match* atau *missmatch*. D adalah matriks dua dimensi berisi hasil perhitungan jarak menggunakan algoritma Smith-Waterman.

D. Algoritma Levenshtein Distance

Algoritma Levenshtein distance banyak digunakan untuk mengukur kesamaan atau kemiripian dua buah kata (string). Jarak Levenshtein diperoleh dengan mencari cara termudah untuk mengubah sebuah string. Algoritma ini mirip dengan algoritma Smith-Waterman. Perbedaan utamanya adalah pada *Levenshtein distance* memiliki perhitungan biaya minimum untuk membuat dua buah string menjadi sama. Sedangkan Smith-Waterman menggunakan *local sequence alignment*, yaitu membuat dua buah string memiliki urutan yang sama dan kemudian menghitung total karakter pada bagian string yang sama.

Secara umum, operasi mengubah yang diperbolehkan untuk keperluan ini adalah [4]:

- memasukkan karakter ke dalam string (*insertions*),
- menghapus sebuah karakter dari suatu string (*deletions*),
- mengganti karakter string dengan karakter lain (*substitutues*).

Persamaan cara menghitung kemiripan dua buah string dapat dilihat pada (3). Variabel $lev_{a,b}(i,j)$ adalah nilai jarak/kemiripan antara karakter ke-*I* pada kata **a** dan karakter ke-*j* pada kata **b**. Sedangkan $i_{a_i \ne b_j}$ merupakan indikator fungsi yang bernilai 0 apabila $a_i = b_j$ dan bernilai 1 apabila sebaliknya.

$$lev_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{jika } \min(i,j) = 0 \\ \min \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+1_{a_i \ne b_j} \end{cases} & \text{sebaliknya} \end{cases} \tag{3}$$

Contoh penggunaan metode *Levenshtein distance* untuk mendeteksi kemiripan dua buah string adalah sebagai berikut. Nilai *Levenshtein Distance* antara dua kata “kitten” dan “sitting” adalah 3, dengan penjelasannya adalah sebagai berikut.

```
 FUNCTION LevenshteinDistance(String S1,String S2)
1. // Initialize M as an empty Array[][]
2. M[0][0] = 0
3. FOR i = 1 to size of String S1
4.    M[i][0] = i
5. FOR j = 1 to size of String S2
6.    M[0][j] = j
7. FOR i = 1 to size of String S1
8.    FOR j = 1 to size of String S2
9.      IF S1[i] = S2[j]
10.        cost = 0
11.     ELSE
12.        cost = 1
13.     M[i][j] = min( M[i-1][j-1]+cost, M[i-1][j]+1, M[i][j-1]+1)
14.     return M[i][j]
```

Gambar 4. Algoritma Levenshtein *distance*

1. **k**itten → **s**itten (substitusi "s" untuk "k")
2. sitt**e**n → sitt**i**n (substitusi "i" untuk "e")
3. sittin → sittin**g** (penambahan "g" pada akhir kata)

Pseudocode dari *Levenshtein Distance* sendiri dapat didefinisikan seperti pada Gambar 4.

E. Hierarchical Clustering

Hiearchical clustering merupakan metode untuk melakukan analisis *cluster* dengan cara membentuk hirarki *cluster*[8]. Strategi yang digunakan di dalam *hierarchical clustering* secara umum terbagi menjadi dua, yaitu:

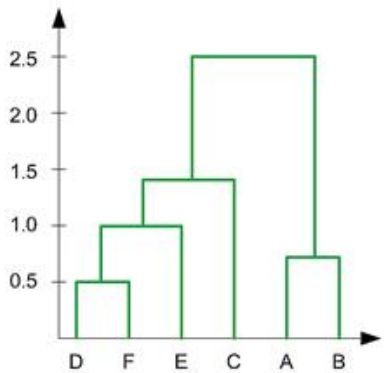
- Aglomeratif
  Aglomeratif merupakan pendekatan "*bottom up*", setiap observasi berawal dari *cluster* itu sendiri, dan pasangan *cluster* akan digabungkan menjadi satu (*merge*) saat naik hirarki.
- Divisif
  Divisif merupakan pendekatan "*top down*", yang berarti bahwa semua observasi diawali dari satu cluster, kemudian proses *split* dilakukan secara rekursif saat proses *clustering* turun satu step.

Secara umum, proses merge dan split ditentukan secara *greedy*, yaitu mengambil keputusan yang dirasa paling optimal pada setiap iterasi. *Hierarchical clustering* biasanya disajikan dalam bentuk grafik yang disebut sebagai dendogram. Grafik ini menggambarkan pembentukan *cluster* sesuai dengan strategi yang dipilih (aglomeratif atau divisif) [9]. Contoh dendogram dapat dilihat pada Gambar 5.

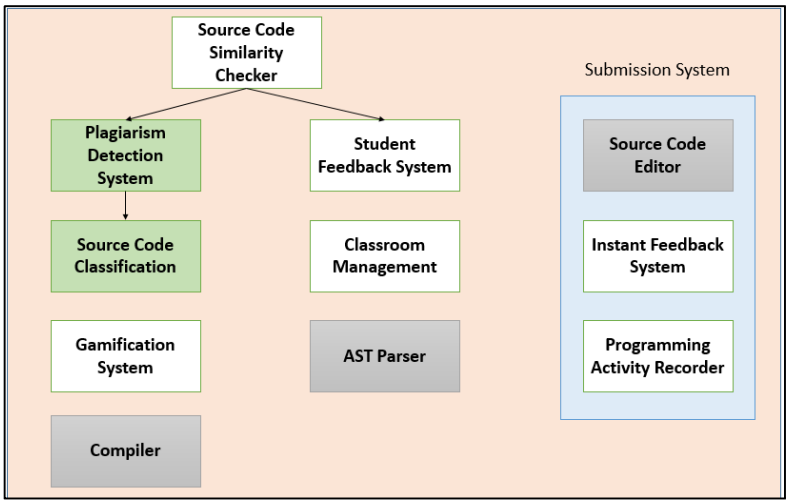
F. Regular Expression

Regular expression atau disebut juga *rational expression* adalah teori bahasa formal dalam ilmu komputer teoritis. *Reguler expression* menggambarkan sebuah urutan karakter yang mendefinisikan pola pencarian [10]. Biasanya pola tersebut akan digunakan di dalam algoritma pencarian untuk menemukan (*find*) dan menemukan-dan-mengganti (*find and replace*) di dalam string. Beberapa notasi dasar dari regular expression antara lain sebagai berikut:

1. | disebut juga alternasi atau pemilihan. Dapat dibaca sebagai "atau".
2. ( ) digunakan untuk mengelompokkan, fungsinya sama persis seperti tanda kurung pada matematika.
3. [ ] mengapit sebuah set karakter. Misal [0-9] cocok dengan angka 0 hingga 9



Gambar 5. Contoh Dendogram



Gambar 6. Arsitektur Sistem E-Learning Pemrograman

4. ? berarti nol atau satu huruf /element di kiri bersifat opsional. Misal colou?r, cocok dengan "color" maupun "colour".
5. * menggambarkan kemunculan karakter sebanyak nol, satu, atau lebih dari satu.
6. ^ dan $ masing-masing dapat disebut sebagai "harus di awal" dan "harus di akhir"
7. {n} karakter atau kelompok huruf di kiri harus berjumlah sebanyak n
8. {min,max} karakter atau kelompok huruf di kiri harus berjumlah minimal sebanyak min dan tidak boleh lebih dari nilai max.

Berikut ini merupakan beberapa contoh pola *regular expression* dan ekspansinya dalam string.

- a|b* menggambarkan {ε, "a", "b", "bb", "bbb", …}, di mana ε merupakan string kosong.
- (a|b)* menggambarkan himpunan semua string yang tidak memiliki simbol selain "a" dan "b", termasuk di dalamnya string kosong: { ε , "a", "b", "aa", "ab", "ba", "bb", "aaa", …}
- ab*(c|ε) menggambarkan himpunan semua string yang dimulai dengan huruf "a", diikuti huruf "b: sejumlah 0 atau lebih dan kemudian diikuti oleh karakter opsional "c": {"a", "ac", "ab", "abc", "abb", "abbc", …}

III. Desain Arsitektur Sistem dan Implementasi

A. *Desain Umum Sistem*

Artikel ini hanya membahas dua modul di dalam sebuah aplikasi *E-learning Pemrograman* yang akan dibangun. Secara umum, arsitektur sistem dapat dilihat pada Gambar 6. Modul yang diimplementasikan dalam artikel ini adalah modul deteksi plagiarisme dan klasifikasi kode program dan sistem umpan balik (*Student Feedback System*). Sedangkan modul-modul lain tidak dibahas di dalam artikel ini.

Modul deteksi plagiarisme dan klasifikasi kode program diimplementasikan dengan memanfaatkan algoritma Levenshtein *distance* untuk menghitung tingkat kemiripan kode program. Algoritma Levenshtein *distance* yang digunakan akan dimodifikasi pada penghitungan elemen yang sama. Hal ini berfungsi agar kode program yang mirip namun hanya berbeda urutan blok statemennya dapat dideteksi sebagai kode yang mirip. Klasifikasi kode program menggunakan algoritma *hierarchical clustering* bertujuan untuk menentukan kelompok kode-kode program berdasarkan persentase kemiripannya.

Model *Student Feedback System* memiliki dua tahap utama. Tahap pertama adalah memproses kode program mahasiswa menjadi sebuah *sequence* yang merupakan representasi bentuk AST kode program. Tahap selanjutnya adalah membandingkan antara dua buah *sequence* kode program, dari dosen dan mahasiswa, dan kemudian menghitung tingkat kemiripannya. Proses perhitungan tingkat kemiripan kode program menggunakan algoritma Smith-Waterman yang telah dimodifikasi. Hasil dari proses pembandingan dua buah kode program ini adalah nilai *similarity*/kemiripan dan sebuah teks yang berisi kode yang sama antara dua buah kode program.

B. *Tahap Praproses*

Tahap praproses bertujuan untuk mengubah data menjadi bentuk yang cocok untuk proses penghitungan kemiripan. Tahap ini mengubah kode program menjadi AST menggunakan ANTLR, kemudian menjadi *sequence* string yang akan dibandingkan. Proses pembandingan *sequence* string ini menggunakan algoritma Levenshtein *distance* pada modul deteksi plagiarisme dan algoritma Smith-Waterman pada modul *Student Feedback System*.

Tabel I. *Grammar* C++ yang Di-*Parsing*

No.	Aturan/*rule* pada *Grammar*	Keterangan pada kode sumber
1	Declaration	Sebuah pendeklarasian suatu kode yang umumnya diakhir dengan tanda ";" (titik koma) pada c++
2	Simpletypespecifier	Tipe data pada suatu variabel (int, char, long, double, dll)
3	Functiondefinition	Pendeklarasian suatu fungsi (void , int , dll)
4	Parametersandqualifiers	Parameter dalam sebuah fungsi
5	Functionbody	Isi suatu fungsi
6	Selectionstatement	Statement if , else if, else, dan switch pada c++
7	Iterationstatement	Statement berupa for, while, do-while pada c++
8	Condition	Sebuah kondisi didalam statement if , else if, else, case, while, for, dan do-while pada c++
9	Classpecifier	Pendeklarasian suatu kelas atau struct pada c++
10	Classkey	Pendeklarasian key sebuah kelas atau struct
11	Accesspecifier	Jenis access specifier pada anggota suatu kelas (*public, private, dan protected*)
12	Basespecifier	Pendeklarasian sebuah kelas turunan pada c++
13	Assignmentexpression	Pendeklarasian sebuah assignment pada program c++
14	Assignmentoperator	Operator yang menghubungkan sebuah *assignment*. Operator terdiri dari (= , += , -= , *= , /= , %=)
15	Unqualifiedid	Penggunaan/deklarasi suatu variabel pada kode program C++
16	Literal	Penggunaan/deklarasi suatu angka pada kode program C++
17	Expressionlist	Isi parameter dari pemanggilan suatu fungsi
18	Postfixexpression	Mendeteksi statement yang diikuti dengan operator (. , -> , -- , ++ , dan ( )
19	Shiftexpression	Mendeteksi penggunaan cout << dan cin >> pada kode program C++
20	Jumpstatement	Mendeteksi penggunaan break
21	Classname	Mendeteksi nama kelas
22	Braceorequalinitializer	Mendeteksi tipe data yang diikuti suatu
23	Ptoperator	Untuk mengidentifikasi adanya deklarasi pointer
24	Primaryexpression	Untuk mengidentifikasi adanya penggunaan this pointer (this →)

1) Parsing ANTLR

Data masukan dari proses ini adalah kode-kode program dalam bahasa C++ yang akan dicek tingkat kemiripannya, baik dari dosen maupun mahasiswa. Kode program tersebut di-*parsing* menggunakan plugin ANTLR menjadi AST dan kemudian diolah menggunakan fungsi listener yang terdapat pada ANTLR *parser*. *Listener* digambarkan seperti *tree-walker* yang berjalan-jalan mengunjungi node dari AST. Listener memiliki dua fungsi utama yaitu *Enter* dan *Exit*. *Enter* digunakan ketika *tree-walker* tersebut mengunjungi suatu node dan *Exit* digunakan ketika *tree-walker* tersebut selesai mengunjungi suatu *node*.

Isi *node* tersebut didapatkan dari *grammar* C++ ANTLR, namun tidak semua segmen kode program akan diambil. Hal ini disebabkan karena perhitungan kemiripan hanya berdasarkan strukturnya. Terdapat 24 aturan pada *grammar* yang akan diambil sebagai elemen dari AST. Aturan-aturan yang diambil disesuaikan dengan tingkat kepentingan di dalam sebuah kode program. Untuk lebih lengkapnya, 24 aturan/*rule* pada *grammar* C++ dapat dilihat pada Tabel I.

2) Pembentukan Sequence

Setelah kode program di-*parsing* oleh ANTLR menjadi AST, maka fungsi *listener* yang dimodifikasi digunakan untuk mengubah bentuk AST menjadi bentuk *sequence*. Proses modifikasi fungsi *listener* teresbut adalah dengan merepresentasikan fungsi-fungsi pada grammar hasil parser tadi ke dalam suatu rangkaian huruf dan memberikan kurung/*parenthesis* di beberapa huruf. Kurung ini digunakan untuk menandai apakah huruf-huruf tersebut berada dalam satu level *tree* atau tidak. Kode berupa rangkaian huruf yang merepresentasikan fungsi-fungsi yang akan di-*parser* pada *grammar* dijelaskan pada Tabel II.

C. Perhitungan Levenshtein Distance

Konsep dasar metode *Levenshtein distance* adalah nilai matriks (*baris, kolom*) diambil dari nilai terkecil (*minimum*) antara nilai matriks(*baris-1, kolom-1*), matriks(*baris-1, kolom*), dan matriks*(baris, kolom-1*). Aturan lain dari algoritma ini adalah apabila elemen pada *sequence* dan level *sequence* kode program pertama dengan elemen pada *sequence* dan level *sequence* pada kode program ke dua bernilai sama, maka matriks pada indeks ke (*baris-1, kolom-1*) ditambah dengan nilai *cost* nol. Begitu pula sebalikanya, nilai *cost*-nya satu. Sedangkan nilai matriks pada indeks ke (*baris, kolom-1*) dan matriks pada indeks (*baris-1, kolom*) selalu ditambah dengan satu. Hasil akhir, berupa nilai jarak antara dua buah *string sequence*, dari metode *Levenshtein distance* ini ada pada baris terakhir dan kolom terakhir matriks.

Tabel II. Pengkodean *rule grammar* C++ menjadi rangkaian huruf

No.	Aturan/*rule* pada *Grammar*	Keterangan pada kode sumber
1	Declaration	**A ( )**
2	Simpletypespecifier	**B : tipe** (tipe merupakan tipe variabel, misal B: int)
3	Functiondefinition	**C ( )**
4	Parametersandqualifiers	**D ( )**
5	Functionbody	**E ( )**
6	Selectionstatement	**H : statement** Misal (**H : if**)
7	Iterationstatement	**I : statement ( )** Misal (**I : for**)
8	Condition	**J (kondisi)** kondisi disini diikuti oleh operotor kondisi, Selain itu kondisi juga dapat diikuti suatu variabel atau literal.
9	Classpecifier	**K ( )**
10	Classkey	Diikuti oleh simbol **K**. Misal : **K:struct** atau **K:class**
11	Accesspecifier	**L : *accessspecifier*** Misal (**L : public**)
12	Basespecifier	**M**
13	Assignmentexpression	Tidak dikodekan.
14	Assignmentoperator	Operator diikuti oleh suatu variabel (**V**) atau angka (literal).
15	Unqualifiedid	**V.** simbol ini dapat diikuti oleh : operator sebuah *assignment* (misal **=V**), parameter pemanggilan fungsi (misal **PV**), kondisi (misal **JV**) statement dari *switch-case* (misal **GV**), simbol-simbol dari postfix expression (misal **VO**)
16	Literal	Angka yang tertera pada kode program. Tidak hanya angka, namun literal juga meliputi string literal. Literal dapat diikuti oleh : operator sebuah *assignment*, parameter pemanggilan fungsi, kondisi, dan statement dari *switch-case*
17	Expressionlist	**P.** simbol ini dapat diikuti oleh : operator sebuah *assignment*,variabel,literal kondisi, statement dari *switch-case*, dan simbol-simbol dari postfix expression
18	Postfixexpression	Mendeteksi *statement* yang diikuti dengan: Access operator (**.**) dan (→) **V.** (misal : Pet.makan) **V→** (misal : Pet → makan) Pemanggilan fungsi **VO** (misal : makan()) Increment **V--** (misal : input--) Decrement **V++** (misal : input++) Pemanggilan fungsi yang diikuti access operator: **V.O** (misal : Pet.makan()) **V→O** (misal : Pet→makan())
19	Shiftexpression	Tidak dikodekan.
20	Jumpstatement	**break** (apabila berada didalam suatu switch-case,disimbolkan menjadi **Gbreak**)
21	Classname	Tidak dikodekan.
22	Braceorequalinitializer	Tidak dikodekan.
23	Ptoperator	Disimbolkan dengan (***** atau **&**)
24	Primaryexpression	**This**

Contoh implementasi penghitungan jarak antara dua buah *string sequence* menggunakan metode *Levenshtein distance* adalah sebagai berikut. Pada matriks Gambar 7, kolom berwarna kuning (*baris, kolom*) didapatkan dari nilai minimum dari tiga buah nilai:

1. kolom berwarna merah (*baris, kolom-1*) + 1
2. kolom berwarna merah muda (*baris-1, kolom*) + 1
3. kolom berwarna hijau (*baris-1, kolom-1*) + 0

Hasil akhir penghitungan matriks *Levenshtein distance* di atas ada pada kolom berwarna biru (baris terakhir, kolom terakhir). Persamaan (4) digunakan untuk menghitung nilai *similarity* dari hasil *Levenshtein distance* yang telah dihitung.

$$similarity = \left(1 - \frac{levenshtein}{\max\left(n_1, n_2\right)}\right) \times 100\% \quad (4)$$

Di mana *levenshtein* merupakan nilai matriks pada baris terakhir dan kolom terakhir pada matriks penghitungan jarak *Levenshtein*. Variabel n_1 merupakan nilai dari panjang *string sequence* kode program

			Sequence kode program 1				
			K:class	C	D	E	B:long
		0	1	2	3	4	5
Sequence kode program 2	K:class	1	0	1	2	3	4
	C	2	1	0	1	2	3
	D	3	2	1	0	1	2
	E	4	3	2	1	0	1
	B:int	5	4	3	2	1	1

Gambar 7. Contoh Perhitungan Levenshtein Distance

pertama dan n_2 merupakan nilai dari panjang *string sequence* kode program kedua. Kelemahan dari metode *Levenshtein distance* dalam mendeteksi plagiarisme antara dua buah kode program adalah tidak mampu mendeteksi isi kode program yang urutan posisi penulisannya ditukar (dibolak-balik). Sehingga untuk mengatasi hal tersebut, diperlukan modifikasi terhadap pengecekan dan penghitungan *similarity* antar dua buah kode sumber.

Modifikasi *similarity* dilakukan untuk mendeteksi susunan/urutan isi kode program yang posisinya ditukar (dibolak-balik). Modifikasi dilakukan dengan cara melakukan pengecekan di setiap isi baris dan kolom matriks. Sehingga apabila terdapat *sequence* yang urutannya ditukar, dapat dideteksi, karena pengecekan *sequence* dilakukan secara menyeluruh pada tiap kolom dan baris. Apabila kedua substring *sequence* dan level *sequence* antara dua buah kode sumber bernilai sama, maka *cost* pada baris atau kolom tersebut bernilai satu. Selanjutnya substring *sequence* yang telah memiliki pasangan yang mirip dengan substring *sequence* lain akan ditandai. Proses penandaan ini digunakan untuk menandai bahwa tidak akan ada substring *sequence* yang memiliki kemiripan lebih dari satu dengan substring *sequence* pembandingnya.

Penghitungan persentase kemiripan modifikasi *similarity* adalah dengan melakukan penjumlahan terhadap baris atau kolom yang memiliki nilai cost satu, selanjutnya hasil penjumlahan tersebut dibagi dengan panjang maksimum antara kedua buah *sequence* kode program yang dibandingkan. Penghitungan persentase kemiripan modifikasi penghitungan *similarity* dituliskan melalui (5).

$$\text{modified-similarity} = \left(\frac{N}{\max\left(n_1, n_2\right)} \right) \times 100\% \tag{5}$$

Di mana N merupakan jumlah *substring sequence* yang sama antara kedua buah *sequence* yang dibandingkan. n_1 merupakan nilai dari panjang *string sequence* kode program ke-1 dan n_2 merupakan nilai dari panjang *string sequence* kode program ke-2.

D. *Hierarchical Clustering*

Hierarchichal clustering merupakan metode yang digunakan dalam proses pengelompokan/klasifikasi kode program berdasarkan tingkat kemiripannya. Konsep dasar *hierarchical clustering* adalah mengambil nilai minimum dari setiap nilai kolom dan baris pada matriks dan menggabungkan kedua elemen baris dan kolom tersebut menjadi satu *cluster*. Matriks yang dimaksudkan tersebut berisi nilai jarak *similarity* antar kode program satu dengan yang lainnya. Matriks tersebut bersifat *mirror*, yang berarti jarak *similarity* antara kode program pertama dan kedua sama dengan *similarity* kode program kedua dan pertama. Jarak *similarity* antar dua buah kode program dihitung melalui (6).

$$dist = 100\% - ms \tag{6}$$

Di mana *ms* merupakan nilai *modified similarity* yang telah dihitung dari (5).

Jika setiap pasangan kode sumber memiliki tingkat kemiripan yang besar maka jaraknya relatif kecil. Sehingga nilai minimum yang diambil dari hasil penghitungan jarak *similarity* tersebut, merupakan nilai maksimum dari hasil penghitungan persentase tingkat kemiripan. Contoh matriks jarak *similarity* antar kode program dapat ditunjukkan pada Gambar 8. Pada gambar tersebut A, B, C, D, E, F merepresentasikan nilai ID dari masing-masing kode program mahasiswa.

Metode *minimum linkage* digunakan untuk mengambil nilai minimum dari isi matriks, yaitu nilai yang paling kecil antara dua buah kode program. *Single linkage* digunakan untuk meng-*update* nilai matriks pada saat penggabungan member dalam proses *clustering*. Proses *clustering* dimulai dari menggabungkan dua buah kode program yang mirip (jarak similarity yang paling kecil), hingga menggabungkan seluruh kode program menjadi satu cluster.

Dist	A	B	C	D	E	F
A	0	0,71	5,66	3,61	4,24	3,2
B	0,71	0	4,95	2,92	3,54	2,5
C	5,66	4,95	0	2,24	1,41	2,5
D	3,61	2,92	2,24	0	1	0,5
E	4,24	3,54	1,41	1	0	1,12
F	3,2	2,5	2,5	0,5	1,12	0

Gambar 8. Contoh matriks jarak *similarity* antar kode program

Alur pengelompokan kode program berdasarkan tingkat kemiripannya menggunakan metode Hierarchical clustering dapat diuraikan seperti berikut:

1. Pilih nilai matriks yang paling kecil dari matriks jarak *similarity* antar kode program yang sudah dihitung sebelumnya.
2. Gabungkan dua buah ID dari nilai matriks yang paling kecil tersebut ke dalam satu *cluster*.
3. Kosongkan isi matriks yang berkaitan dengan kedua ID tersebut.
4. Matriks yang kosong tersebut dihitung nilainya menggunakan metode *Single linkage*. Metode ini mencari nilai minimum antara anggota-anggota *cluster* dengan anggota di luar *cluster*.
5. Selanjutnya cari lagi nilai minimum matriks dengan metode *minimum linkage*. Kemudian gabungkan menjadi satu *cluster* seperti pada langkah ke 2.
6. Ulangi langkah di atas, hingga semua matriks bergabung menjadi satu *cluster*.

E. Penghitungan Kemiripan Kode Program dengan Menggunakan Smith-Waterman dan Backtracking

Smith-Waterman merupakan metode yang digunakan untuk menghitung nilai kemiripan antara dua buah *sequence* (sekumpulan karakter). Data masukan dalam proses ini adalah dua buah *sequence* yang merupakan hasil dari tahap praproses sebelumnya. Metode *Smith-Waterman* membutuhkan sedikit modifikasi untuk menyelesaikan beberapa *case* dalam pendeteksian kemiripan dua buah kode sumber, karena metode ini hanya mendeteksi secara berurutan tidak dapat mendeteksi kode yang peletakannya dibalik.

Pada metode *Smith-Waterman* ini apabila tiap huruf pada dua buah *sequence* adalah sama dan memiliki level yang sama pula, maka akan disebut *match* yang mempunyai nilai pada perhitungan *Smith-Waterman* tersebut adalah +2. Apabila tidak sama atau *mismatch*, *cost*-nya bernilai -1. Nilai tersebut akan ditambahkan dengan nilai matriks 2 dimensi hasil yang ada pada indeks (*i*-1, *j*-1). Sedangkan ada sebuah nilai yang disebut W (bernilai -1) yang ditambahkan dengan nilai matriks 2 dimensi hasil yang ada pada indeks (*i*-1, *j*) atau (*i, j*-1). Lalu diambil jumlah nilai yang paling besar untuk mengisi matriks hasil serta disimpan arahnya pada matriks *directions*. Arahnya disimbolkan untuk atas adalah 3, kiri adalah 2, dan miring adalah 1 jika *match*, dan 9 jika *mismatch*.

Sedangkan modifikasi dilakukan untuk melakukan deteksi terhadap susunan isi kode program yang posisinya ditukar (dibolak-balik). Modifikasi yang dilakukan adalah mencatat semua nilai yang *match* yang akan dimasukkan ke dalam array 1 dimensi. Panjang dimensi adalah baris atau kolom yang paling panjang. Selain itu, indeks nilai *match* akan disimpan pada *arraylist*. Hasil persentase *similarity* menggunakan metode Smith-Waterman yang tidak dimodifikasi maupun yang dimodifikasi akan dibandingkan dan dicari nilai akurasi yang lebih besar.

Backtracking digunakan untuk mengetahui kode mana saja yang mirip dari dua buah source code dan menghitung nilai kemiripan dua buah kode program. Baik metode Smith-Waterman dan Smith-Waterman yang dimodifikasi menggunakan rumus yang sama untuk menghitung kemiripan dua buah kode sumber. Rumus menghitung kemiripan dua buah source code ditunjukkan pada (7).

$$sim = \frac{a}{\max(i, j)} \times 100\% \tag{7}$$

di mana *a* merupakan banyaknya kode program yang mirip.

Pada metode Smith-Waterman, *backtracking* dimulai dari indeks matriks hasil ke i=panjang baris dan j = panjang kolom. *Backtracking* mengikuti isi matriks directions, jika benilai 2 maka ke kiri, jika bernilai 3 maka ke atas, dan jika bernilai 1 atau 9 maka ke kiri-atas (serong). *Backtracking* berhenti jika indeks i atau j bernilai 0. Sedangkan *backtracking* modifikasi mencocokkan indeks baris atau kolom dengan isi *arraylist* yang berisi indeks nilai *match*. Selama indeks *match* belum pernah digunakan, maka indeks tersebut akan dipakai sebagai indeks yang memiliki kemiripan dua buah buah kode sumber.

IV. PENGUJIAN DAN PEMBAHASAN

A. *Dataset yang Digunakan*

1) *Dataset Modul Deteksi Plagiarisme*

Data yang digunakan pada modul ini berupa kode program mahasiswa dalam satu kelas. Kode program tersebut merupakan hasil pengerjaan tugas pemrograman yang diberikan oleh dosen Teknik Informatika ITS. Data yang diambil terdiri dari dua jenis kode program mahasiswa dalam satu kelas yang mengambil mata kuliah Pemrograman Berorientasi Objek (PBO). Dataset jenis pertama berjumlah 38 buah kode program dari 38 mahasiswa dalam satu kelas. Dataset jenis ke 2 berjumlah 21 buah kode program dari 21 mahasiswa dalam satu kelas. Pada dataset jenis kedua terdapat dua jenis file yaitu 21 file *header* (.h) dan 21 file yang berekstensi .cpp. Sehingga masing-masing file akan dihitung perfomanya pada tahap uji coba.

2) *Dataset Modul Student Feedback System*

Data yang digunakan pada modul ini berasal dari tugas mata kuliah Pemrograman Berorientasi Objek (PBO) Kelas C Tahun Ajaran 2013/2014 di Jurusan Teknik Informatika ITS. Dataset yang digunakan adalah tiga jenis dataset, yaitu dataset class Invoice, class Account, serta dataset kode sumber yang tidak mirip. Jumlah dataset pertama adalah 31 buah kode program mahasiswa, jumlah dataset kedua adalah 31 buah mahasiswa, dan jumlah dataset ketiga adalah 9 mahasiswa.

B. *Proses Pengujian yang Dilakukan*

1) *Pengujian Modul Deteksi Plagiarisme*

Pengujian dilakukan dengan menilai akurasi pada penghitungan nilai kemiripan dua buah kode program dan akurasi pengelompokan kode program dengan menggunakan jumlah cluster hasil penghitungan standar deviasi. Pengujian dilakukan menggunakan dua buah dataset yang telah dijelaskan sebelumnya.

Pengujian pada penilaian akurasi penghitungan nilai kemiripan kode program dilakukan secara manual dengan menggunakan *ground-truth* yang memiliki 3 buah parameter (label) yaitu Tidak Mirip (0%-35%), Setengah mirip (36%-65%), dan Mirip (66%-100%). Penilaian dilakukan pada kedua metode yang digunakan untuk menghitung nilai kemiripan antar kode program, yaitu metode *Levenshtein distance* dan modifikasi *similarity*.

Pengujian pada penilaian akurasi pengelompokan kode program dilakukan dengan memberi label *True* dan *False* pada masing-masing anggota *cluster*. Penilaian akurasi pengelompokan kode program, pada jumlah cluster hasil penghitungan standar deviasi, dilakukan secara manual dengan menilai apakah kode program-kode program yang terdapat dalam satu *cluster* memang mirip satu sama lain dan apakah kode program yang menjadi satu-satunya member pada suatu *cluster* benar-benar tidak mirip dengan kode program manapun. Apabila sesuai maka nilai nya *True*, apabila tidak sesuai maka nilainya *False*. Kemudian dihitung rata-rata dari penilaian tersebut untuk mendapatkan nilai akurasinya.

2) *Pengujian Modul Student Feedback System*

Pengujian dilakukan dengan menilai akurasi pada penghitungan nilai kemiripan dua buah kode program. Uji coba dilakukan sebanyak tiga kali menggunakan tiga buah dataset yang telah dijelaskan sebelumnya. Akurasi pada perhitungan kemiripan kode program dilakukan dengan menilai kemiripan dua buah kode program secara manual. Penilaian dilakukan dengan memberikan 2 jenis nilai *ground truth*, yaitu Tidak Mirip (persentase kemiripan antara 0-45%), dan Mirip (46%-100%).

Skenario pengujian kedua adalah penampilan rekomendasi kode program yang mirip milik mahasiswa dengan milik dosen. Sistem akan menampilkan kode program mahasiswa dan dosen secara berdampingan. Pada tampilan tersebut, kode program yang sama/sesuai dengan kode program dosen akan diberi tanda khusus.

C. *Hasil dan Evaluasi*

1) *Hasil dan Evaluasi Modul Deteksi Plagiarisme*

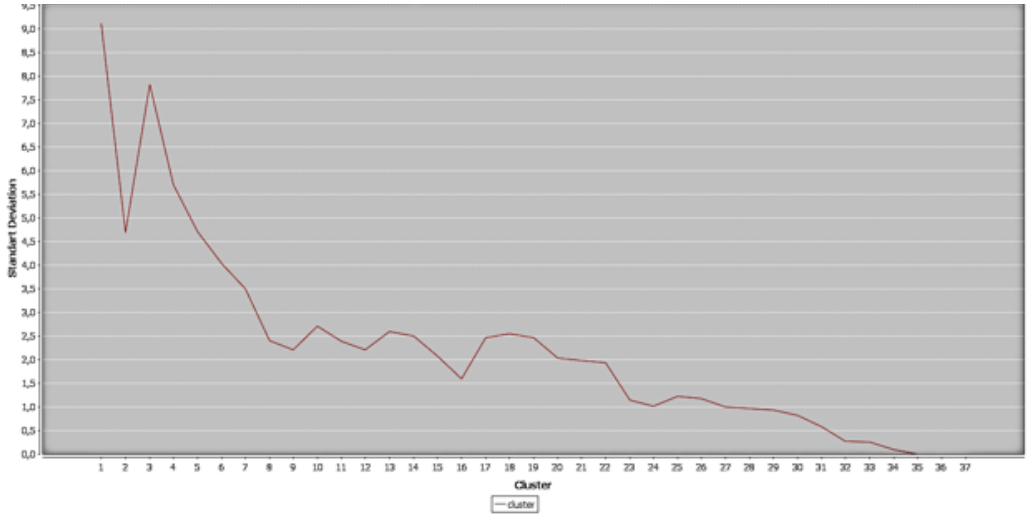
Skenario uji coba 1 adalah penghitungan akurasi sistem pendeteksi plagiarisme kode program dengan menggunakan dataset jenis pertama yaitu dataset yang memiliki tingkat kemiripan antar kode program yang relatif kecil. Uji coba dilakukan dengan menghitung akurasi pada penghitungan nilai similarity antar kode program, pengelompokan kode program berdasarkan tingkat kemiripannya, dan penentuan jumlah cluster terbaik yang akan dipilih. Terdapat 38 kode program pada dataset jenis pertama yang memiliki total 703 nilai *similarity*.

Tabel III. Hasil Akurasi Penghitungan Kemiripan Kode Program pada Pengujian 1 dengan 3 Parameter (Levenshtein Distance)

Levenshtein distance		*Predicted*		
		Mirip	Setengah mirip	Tidak mirip
Actual	Mirip	0	0	0
	Setengah mirip	0	10	28
	Tidak mirip	0	3	662
Akurasi		**95,59%**		

Tabel IV. Hasil Akurasi Penghitungan Kemiripan Kode Program pada Pengujian 1 dengan 3 Parameter (Levenshtein modifikasi)

Modifikasi *similarity*		*Predicted*		
		Mirip	Setengah mirip	Tidak mirip
Actual	Mirip	0	0	0
	Setengah mirip	0	38	0
	Tidak mirip	0	75	590
Akurasi		**89,33%**		



Gambar 9. Evaluasi Standar Deviasi pada Hierarchical Clustering Pengujian 1

Berdasarkan hasil yang ditunjukkan pada akurasi penghitungan kemiripan kode program dengan 3 parameter pada Tabel III dan Tabel IV, metode *Levenshtein distance* memiliki akurasi lebih tinggi dibanding *Modifikasi similarity* yaitu mencapai **95,59%.** Sedangkan apabila menggunakan metode *modifikasi similarity* akurasinya mencapai 89,33%. Hal ini dikarenakan data masukan pada uji coba pertama memiliki tingkat kemiripan yang relatif kecil.

Nilai yang digunakan dalam proses pengelompokan kode program adalah nilai kemiripan yang dihasilkan pada metode *Modifikasi similarity*. Pengelompokan kode program berdasarkan tingkat kemiripannya dilakukan menggunakan metode *Hierarchical clustering.* Sedangkan pemilihan jumlah cluster yang tepat dilakukan dengan menghitung standar deviasi masing-masing *cluster*. Proses penilaian akurasi pengelompokan kode program dan pemilihan jumlah cluster yang akan diambil dilakukan secara manual, apakah jumlah *cluster* dan pengelompokan kode programnya telah sesuai atau belum. Hasil keluaran berupa grafik dari standar deviasi masing-masing jumlah *cluster* pada uji coba 1 ditunjukkan pada Gambar 9.

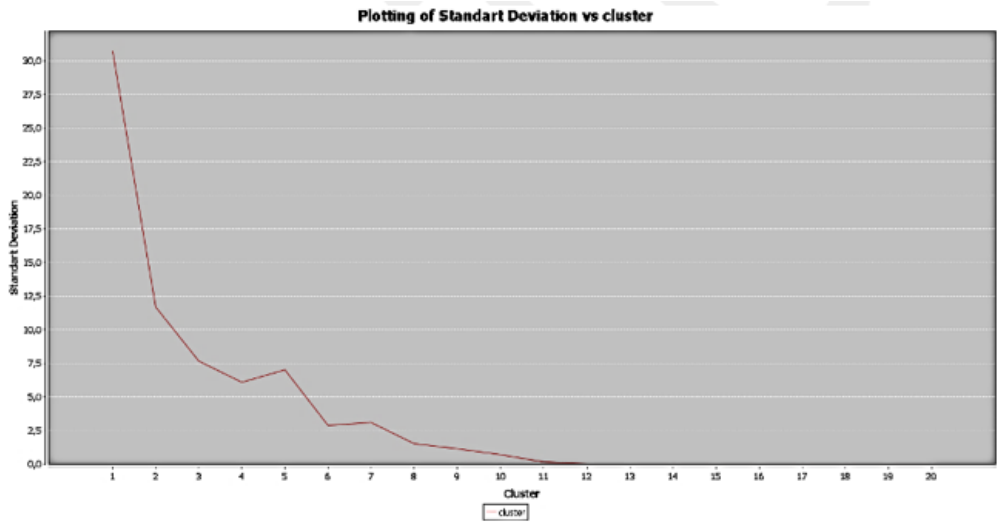
Skenario uji coba 2 adalah penghitungan akurasi sistem pendeteksi plagiarisme kode program dengan menggunakan dataset jenis kedua berupa file .cpp, yaitu dataset yang memiliki tingkat kemiripan antar kode program yang relatif besar. Uji coba dilakukan dengan menghitung akurasi pada penghitungan nilai similarity antar kode program, pengelompokan kode program berdasarkan tingkat kemiripannya, dan penentuan jumlah cluster terbaik yang akan dipilih. Terdapat 38 kode program pada dataset jenis kedua(berekstensi .cpp) yang memiliki total 210 nilai *similarity*. Dari total 210 nilai itu dihitung akurasinya dengan cara menilai secara manual, apakah kedua kode program termasuk dalam kategori tidak mirip atau mirip. Apabila nilai persentase kemiripan antar kedua kode program sesuai dengan kategori pada hasil

Tabel V. Hasil Akurasi Penghitungan Kemiripan Kode Program pada Pengujian 2 dengan 3 Parameter (Levenshtein Distance)

Levenshtein distance		*Predicted*		
		Mirip	Setengah mirip	Tidak mirip
Actual	Mirip	33	0	12
	Setengah mirip	0	39	40
	Tidak mirip	0	3	83
Akurasi		**73,81%**		

Tabel VI. Hasil Akurasi Penghitungan Kemiripan Kode Program pada Pengujian 2 dengan 3 Parameter (Levenshtein Modifikasi)

Modifikasi similarity		*Predicted*		
		Mirip	Setengah mirip	Tidak mirip
Actual	Mirip	45	0	0
	Setengah mirip	4	72	3
	Tidak mirip	0	24	62
Akurasi		**85,24%**		



Gambar 10. Evaluasi Standar Deviasi pada Hierarchical Clustering Pengujian 2

penilaian manual, maka masuk dalam kolom TP (*true positive*) atau TN (*True Negative*).

Berdasarkan hasil yang ditunjukkan pada akurasi penghitungan kemiripan kode program dengan **3** parameter pada Tabel V dan Tabel VI, metode *Modifikasi similarity* memiliki akurasi lebih tinggi dibanding metode *Levenshtein distance* yaitu mencapai **85,24%.** Sedangkan apabila menggunakan metode *Levenshtein distance* akurasinya mencapai 73,81%.

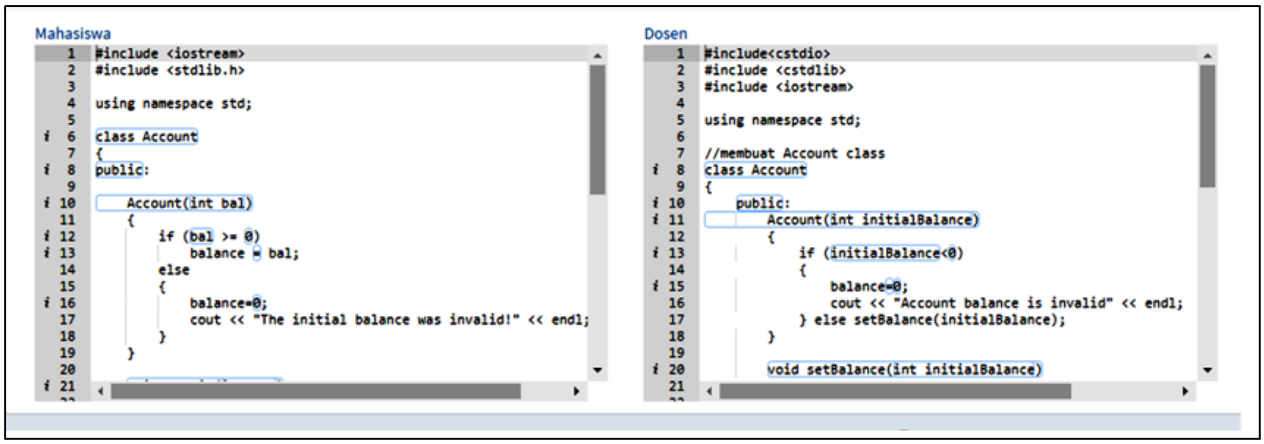
Oleh karena itu nilai yang digunakan dalam proses pengelompokan kode program adalah nilai kemiripan yang dihasilkan pada metode *Modifikasi similarity*. Pengelompokan kode program berdasarkan tingkat kemiripannya dilakukan menggunakan metode *Hierarchical clustering.* Sedangkan pemilihan jumlah cluster yang tepat dilakukan dengan menghitung standar deviasi masing-masing *cluster*. Proses penilaian akurasi pengelompokan kode program dan pemilihan jumlah cluster yang akan diambil dilakukan secara manual, apakah jumlah *cluster* dan pengelompokan kode programnya telah sesuai atau belum. Hasil keluaran berupa grafik dari standar deviasi masing-masing jumlah *cluster* pada uji coba 2 digambarkan pada Gambar 10. Dari hasil grafik standar deviasi tersebut terlihat bahwa jumlah *cluster* ideal dari dataset ini adalah sebanyak 6 *cluster*.

2) Hasil dan Evaluasi Modul Student Feedback System

Pengujian pada modul ini dilakukan dengan cara melihat hasil transformasi AST ke dalam *sequence*. *Sequence* ini nantinya akan dibandingkan dengan *sequence* lain dan akan dihitung persentase kemiripannya berdasarkan algoritma Smith-Waterman. Contoh *sequence* hasil transformasi dari kode program dosen dan mahasiswa dapat dilihat pada Tabel VII. Sedangkan hasil keluaran dari sistem *e-learning* pemrograman yang

Tabel VII. Hasil Teks Kemiripan Kode Program Dosen dan Mahasiswa Menggunakan Metode Smith-Waterman Modifikasi

Kode Sumber Mahasiswa
6 0 12 code: class Account||-:-||8 0 5 code: public||-:-||10 0 16 code: Account Account(intbal)||-:-||10 9 15 code: Parameter : int bal||-:-||12 6 8 code: bal||-:-||12 13 13 code: 0||-:-||13 11 11 code: =V ||-:-||16 11 11 code: =0||-:-||21 1 20 code: void credit(intbal)||-:-||21 13 19 code: Parameter : int bal||-:-||26 12 18 code: Parameter : int bal||-:-||28 5 7 code: bal||-:-||28 12 18 code: balance||-:-||29 10 11 code: -=V ||-:-||29 12 14 code: -=bal ||-:-||23 9 10 code: +=V ||-:-||23 11 13 code: +=bal ||-:-||34 1 16 code: int getBalance()||-:-||36 2 16 code: returnbalance||-:-||39 0 6 code: private||-:-||40 1 11 code: int balance||-:-||43 0 9 code: int main()||-:-||44 1 9 code: return0
Kode Sumber Dosen
8 0 12 code: class Account||-:-||10 1 6 code: public||-:-||11 0 28 code: Account Account(intinitialBalance)||-:-||11 10 27 code: Parameter : int initialBalance||-:-||13 12 25 code: initialBalance||-:-||13 27 27 code: 0||-:-||15 15 15 code: =V ||-:-||15 16 16 code: =0||-:-||20 2 36 code: void setBalance(intinitialBalance)||-:-||20 18 35 code: Parameter : int initialBalance||-:-||25 12 26 code: Parameter : int debitAmount||-:-||27 9 19 code: debitAmount||-:-||27 21 27 code: balance||-:-||28 17 18 code: -=V ||-:-||28 19 29 code: -=debitAmount ||-:-||33 12 13 code: +=V ||-:-||33 14 25 code: +=creditAmount ||-:-||36 2 17 code: int getBalance()||-:-||38 3 17 code: returnbalance||-:-||41 1 7 code: private||-:-||42 4 14 code: int balance||-:-||45 0 9 code: int main()||-:-||47 0 8 code: return0



Gambar 11. Tampilan Umpan Balik (*feedback*) pada Bagian Kode Program yang Mirip Milik Mahasiswa dan Dosen

dapat menampilkan baris kode program yang mirip antara mahasiswa dan dosen ditunjukkan pada Gambar 11.

Data keluaran menggunakan metode Smith-Waterman maupun metode yang sudah dimodifikasi mempunyai kode yang sama. "6 0 12 code: class Account||-:-||" Angka pertama adalah baris line *source code* (angka 6), angka kedua adalah indeks kolom mulai pada *source code* (angka 0), angka ketiga adalah indeks kolom berhenti pada *source code* (angka 12), dan tulisan setelahnya "code:" adalah kode *sequence* ataupun hasil isi *source* code, sedangkan "||-:-||" sebagai pembeda antara kode kemiripan yang satu dengan lainnya. Pada kode sequence kata JV <nama variabel/literal> berarti berarti variabel tersebut berada pada kondisi, sedangkan =V atau =<nama variabel/literal> atau +=<nama variabel> berarti variabel tersebut merupakan variabel pada *assignment.*

V. KESIMPULAN

Penelitian ini bertujuan untuk mendeteksi plagiarisme kode program dalam satu kelas dan mengelompokkan berdasarkan tingkat kemiripannya. Kakas bantu ANTLR dan aturan grammar C++ digunakan untuk memparsing kode program menjadi bentuk *Abstract Syntax Tree* (AST). AST tersebut kemudian diubah menjadi bentuk sequence menggunakan fungsi *listener*. Fungsi Listener dibuat dengan mengkodekan rule pada grammar C++ menjadi rangkaian huruf yang disertai dengan parenthesis sebagai penanda level antara subsequence.

Metode yang digunakan untuk menghitung nilai kemiripan antara dua buah kode sumber adalah metode Levenshtein *distance* dan modifikasi *similarity*. Metode modifikasi similarity digunakan untuk mendeteksi isi kode program yang urutan posisinya ditukar, yang tidak dapat dideteksi oleh metode Levenshtein *distance*. Sedangkan metode yang digunakan untuk mengelompokkan kode program berdasarkan tingkat kemiripannya

adalah metode *hierarchical clustering*. Penghitungan standar deviasi digunakan untuk menentukan jumlah cluster terbaik yang akan diambil.

Hasil pengujian menunjukkan bahwa metode modifikasi *similarity* terbukti lebih akurat dalam mendeteksi beberapa case dalam tindakan plagiarisme kode program, dibandingkan dengan metode Levenshtein *Distance*. Metode *hierarchichal clustering* cukup efektif dalam mengelompokkan kode program berdasarkan tingkat kemiripannya. Pada modul *Student Feedback System* digunakan algoritma Smith-Waterman untuk mengecek kemiripan antara kode program mahasiswa dan dosen. Berdasarkan hasil pengujian dapat disimpulkan bahwa proses transformasi AST ke dalam *sequence* telah berhasil dilakukan. Proses perhitungan kemiripan menggunakan algoritma Smith-Waterman juga dapat dilakukan dengan baik dan dapat digunakan untuk menentukan tingkat kemiripan kode program. Sistem juga telah berhasil menampilkan kode program yang mirip ke dalam tampilan editor.

Ada beberapa deklarasi dan penggunaan syntax pada C++ yang belum bisa dibaca oleh sistem, dikarenakan belum adanya pengembangan lebih lanjut terhadap isi grammar C++ ANTLR. Antara lain penggunaan *array* dan isi kode program berupa *else* masih belum bisa berdiri sendiri menjadi level yang berbeda. Grammar C++ yang telah disediakan ANTLR memiliki cakupan yang terlalu luas dalam representasi bentuk AST-nya. Sehingga diperlukan banyak modifikasi pada *grammar* untuk membaca beberapa deklarasi dan penggunaan *syntax* pada C++ yang belum bisa dibaca oleh sistem.

DAFTAR PUSTAKA

[1] "Sphere Online Judge (SPOJ)," *spoj.com*. [Daring]. Tersedia pada: http://www.spoj.com/. [Diakses: 24-Jan-2017].

[2] X. Li dan X. J. Zhong, "The Source Code Plagiarism Detection Using AST," in *2010 International Symposium on Intelligence Information Processing and Trusted Computing*, 2010, hal. 406–408.

[3] E. Ayguade, J. J. Navarro, dan D. Jimenez-Gonzalez, "Smith-Waterman Algorithm." [Daring]. Tersedia pada: http://docencia.ac.upc.edu/master/AMPP/slides/ampp_sw_presentation.pdf. [Diakses: 23-Des-2016].

[4] V. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Cybernetics and Control Theory*, vol. 10, no. 8, hal. 707–710, Feb 1966.

[5] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. The Pragmatic Programmers, 2013.

[6] T. Parr, "Parse-Tree Listeners and Visitor," in *The Definitive ANTLR 4 Reference, 2nd Edition*, Pragmatic Bookshelf, 2013, hal. 17–20.

[7] "antlr/grammars-v4," *GitHub*. [Daring]. Tersedia pada: https://github.com/antlr/grammars-v4. [Diakses: 23-Jan-2017].

[8] L. Rokach dan O. Maimon, "Clustering Methods," in *Data Mining and Knowledge Discovery Handbook*, O. Maimon dan L. Rokach, Ed. Springer US, 2005, hal. 321–352.

[9] "Hierarchical Clustering Tutorial: Numerical example." [Daring]. Tersedia pada: http://people.revoledu.com/kardi/tutorial/Clustering/Numerical%20Example.htm. [Diakses: 23-Jan-2017].

[10] J. E. F. Friedl, *Mastering Regular Expressions*, 3rd ed. O'Reilly Media, 2006.

[11] L. ping Zhang dan D. sheng Liu, "AST-based multi-language plagiarism detection method," in *Software Engineering and Service Science (ICSESS), 2013 4th IEEE International Conference on*, 2013, hal. 738–742.